Solving Reverse $k$-Nearest Queries on Road Networks with the GPU *

Yago Diez † Marta Fort ‡ J. Antoni Sellarès §

Abstract

We present a GPU-based approach for computing discretized distance functions on road networks. As applications, we provide algorithms for computing discrete Order-$k$ Nearest Neighbor diagrams and for approximately solving (Bichromatic) Reverse $k$-Nearest Neighbor queries on road networks. Finally, we present experimental results obtained with the implementation of our algorithms that demonstrate the effectiveness and efficiency of our approach.

1 Introduction

A Road Network Database allows to efficiently store and query objects in road networks. Static objects are represented by points of interest located on the network (hotels or gas stations) and moving objects are represented as points running along the network (cars or pedestrians). The network distance between objects on the road network (car and gas station), defined as the cost of the shortest path between them, depends on the connectivity and weights of the underlying network.

Computing shortest paths, and consequently network distances, is a fundamental combinatorial optimization problem with important applications in various domains, like Geographic Information Systems, Location-Based Services, Navigation Systems, Mobile Computing Systems, Data Clustering, etc. The computation of network distances often arises as a subroutine in the solution of many proximity problems related the determination of the influence of an object on other objects.

1.1 Preliminaries

We model a road network as an undirected weighted graph $N(V,E,W)$. The set $V$, $n = |V|$, is a set of vertices representing road intersections (with degree above 2), terminal points (with degree 1) and shape points (with degree 2). Shape points are neither intersection nor terminal points that are interpolated to create sequences of arbitrarily small linear parts of the road network. The spatial position of each vertex in $V$ with respect to a reference coordinate system is also given. The set $E$, $m = |E|$, is a set of edges representing road segments, each connecting two vertices. $W : E \to \mathbb{R}^+$ associates each edge $e$ with a positive weight $w(e)$, that may represent, for example, the Euclidean length of $e$, denoted $|e|$, or the time, toll, energy consumption, etc., required to travel between the two endpoints of $e$. A portion $\bar{e}$ of and edge $e$ connecting to points of $e$ is called subedge. We define the weight $w(\bar{e})$ of the subedge $\bar{e}$ as $w(\bar{e}) = (|\bar{e}|/|e|)w(e)$. In this paper we consider static road networks with a fixed weight for each edge. The set of all the points of the network $N(V,E,W)$, vertices and points on edges, is denoted by $N$. Although in the worst case a graph could be dense ($m = O(n^2)$), in general, graphs modelling networks have constant degree and consequently are sparse ($m = \Theta(n)$).

We want to note that for simplicity we have modelled a road network as an undirected graph, but that our methods can be easily applied to road networks modelled as a directed graph, in which one-way roads or roads where the weight is different for the two directions are allowed.

---

*Mart&and J. Antoni Sellarès are partially supported by the Spanish MEC grant TIN2007-67982-C02-02.

†Institut d'Informàtica i Aplicacions, Universitat de Girona, Spain, ydiez@ima.udg.edu

‡Institut d'Informàtica i Aplicacions, Universitat de Girona, Spain, mfort@ima.udg.edu

§Institut d'Informàtica i Aplicacions, Universitat de Girona, Spain, sellares@ima.udg.edu
A path $\pi(p, q)$ between two points $p$ and $q$ located on edge $(v_0, v_1)$ and $q$ located on edge $(v_l, v_{l+1})$, is a sequence $[p, v_1, \ldots, v_l, q]$, where $v_1, \ldots, v_l \in V$ and $(v_{i-1}, v_i) \in E$, $1 \leq i \leq l$. The cost of the path $\pi(p, q)$ is defined by:

$$||\pi(p, q)|| = w(pv_1) + \sum_{2<i\leq l} w(v_{i-1}v_i) + w(v_lq).$$

The path of least cost connecting $p$ and $q$ is called shortest path between $p$ and $q$. The cost of the shortest path is called the network distance between $p$ and $q$, denoted $d(p, q)$. In the following, for simplicity, we use the term distance to refer to the network distance. The distance function defined by a site $p$ on $\mathcal{N}$ is a function $d_p$ such that for any point $q \in \mathcal{N}$, $d_p(q)$ is the distance $d(p, q)$ between $p$ and $q$.

The Order-$k$ Nearest Neighbor diagram of a set of points of interest, called sites, on a network is the partition of the network into $k$-th Nearest regions, where the region associated with a site is the set of points on the network for which the site ranks number $k$ in the ascending order of the sites by network distance. A Reverse $k$-Nearest Neighbor query returns all sites that have a given query point among their $k$-nearest neighbors. In the bichromatic case there are two distinct site types, and a Bichromatic Reverse $k$-Nearest Neighbor query returns all sites of one type that have a given query site of the other type among their $k$-nearest neighbors.

1.2 Graphics Hardware

The graphics pipeline is divided into several stages. The input is a list of 3D geometric primitives expressed as vertices defining points, lines, etc. with attributes associated such as color or texture coordinates, among others. The output is an image in the frame buffer, a collection of several hardware buffers corresponding to 2D grids whose cells are called pixels. A pixel can store a depth value in the depth buffer and up to four values in the color buffer, namely the color channels R, G, B, and alpha. In the first stage of the pipeline, per-vertex operations take place, each input vertex is transformed from 3D coordinates to window coordinates. Next stage is rasterization, when it finishes we obtain a fragment, with its associated attributes, for each pixel location covered by a geometric primitive. Fragment attributes are obtained from the attributes associated to the vertices by linear interpolation. The third stage, the fragment stage, computes the color for each pixel in the frame buffer, according to the results of a series of per-fragment operations and tests, such as the depth test. This output can be transferred to the CPU or stored in a texture, a 2D array of pixels, and then used in additional computations. The programmable parts of the graphics pipeline are the vertex, geometric and fragment shaders which are used to change the vertex attributes, create new vertices or change fragment attributes. Fragment shaders can read in and modify texture contents in a per-pixel fashion, so that the same instruction call is executed for all pixels.

1.3 Related Work

In [2], a multi-pass GPU-based algorithm to efficiently compute approximated Order-$k$ Nearest Neighbor diagrams for a set of points in the Euclidean plane is presented. Algorithms for solving (Bichromatic) Reverse $k$-Nearest Neighbor queries in large graphs are presented in [6]. The majority of works on queries present solutions that mainly focus on spatial data structures (R-trees, etc) and branch and bound pruning techniques that use location and connectivity information to guide the search.

1.4 Our Contribution

By working toward practical solutions, we present an algorithm that takes advantage of the capabilities of the GPU to compute distance functions on road networks. The discretization of the network motivated by the use of the GPU makes solutions approximate by necessity. However this is reasonable since, in practice, the underlying network is often known only approximately. Moreover, in real-world settings, for example it is useful to know not only the nearest gas station, but also other stations that are "close enough" to that nearest one. As applications, we provide algorithms to compute discrete Order-$k$ Nearest Neighbor diagrams and for approximately solving (Bichromatic) Reverse $k$-Nearest Neighbor queries. Finally, we present experimental results that demonstrate the effectiveness and efficiency of the proposed approach.
2 Distance Function Computation

We compute the distance from a given site \( p \) on \( \mathcal{N} \) to all vertices of \( V \) by using the classical Dijkstra’s algorithm [1]. Dijkstra’s algorithm computes the shortest path from site \( p \) to all vertices of \( V \) by incrementally growing a tree of shortest paths from \( p \) out to the most distant vertices. Dijkstra’s algorithm for sparse graphs, like road networks, has \( O(n \log n) \) asymptotic time complexity for an undirected weighted graph with \( n \) vertices [5]. In order to reduce computation times several speed-up techniques have been developed during the last years [3, 5].

Now we face the most general problem of computing the distance from a given site \( p \) to any point on \( \mathcal{N} \). Since the continuous nature of the problem makes it difficult to efficiently compute this distance, we propose an alternative GPU solution based on a discretization (approximation) process that allows us to explicitly store the distance function on a compact way.

Along the paper several time complexity analysis are presented using the following notation. We denote by \( p \) the time needed to render and color a fragment, which is much smaller than that needed to read a pixel, denoted by \( t \), or the time needed to transfer information from the CPU to the GPU, denoted by \( T \). The time needed to render a rectangle covering \( H \times W \) pixels is denoted by the constant \( HWp \), and finally we denote \( O(Np) \) the time needed to render the edges defining the network \( \mathcal{N} \).

2.1 Road Network Parameterization

Let \( R \) be a rectangular grid of the \( xy \)-plane of size \( W \times H \). Consider a mapping \( \tau \) that maps each edge \( e \) of the road network \( \mathcal{N} \) to a row segment of \( R \) represented by the centers of its cells (see Fig. 1a). The mapping optimally packs the mapped edges into \( R \) so that: a) the mappings of two different edges of \( \mathcal{N} \) do not overlap in \( R \); b) the mapping of all edges of \( \mathcal{N} \) covers “almost” all grid cells of \( R \), it is to say, the number of row cells covered by \( \tau(e) \) “approximately” equals \( WH|e|/ \sum_{e \in E} |e'|. \) This way we obtain a discrete parametrization of the road network \( \mathcal{N} \) meaning that each cell in the grid map represents a unique location on the road network. Notice that the converse is not true, a vertex of degree \( d \) is mapped to \( d \) different positions of \( R \). Finally, we denote \( \tau(e, v) \) the mapping of vertex \( v \) when considering it as part of edge \( e \).

2.2 Distance Function Discretization

To compute a discretization of the distance function \( d_p \), defined by site \( p \) on \( \mathcal{N} \), the region \( R \) is represented by using the depth buffer. Consequently, \( R \) is thought as a grid of size \( W \times H \) where distance values, normalized into the interval \([0, 1]\), are stored (see Fig. 1b). Discrete distances are computed by using Dijkstra’s algorithm and the GPU.

We compute the discretized distance function \( d_p \) during Dijkstra’s algorithm. Initially all the points in \( R \) are assumed to be at distance one (the maximal depth value) from the source \( p \). During Dijkstra’s algorithm we compute the distances from \( p \) to the vertices \( v \) of \( \mathcal{N} \) and to the other points of \( \mathcal{N} \) simultaneously. When the distance from \( p \) to a vertex \( v \) is updated by Dijkstra’s algorithm, the distances to the edges incident to \( v \) are also recalculated and updated by using graphics hardware.

For every edge \( e = vw \) incident to \( v \) we render \( \tau(e) \) on \( R \) in order to recompute and update the distances to the points of \( e \). When segment \( \tau(e) \) is rendered we use texture coordinates to associate distance 0 to the endpoint \( \tau(e, v) \) and the distance from \( v \) to \( w \) to \( \tau(e, w) \). The current distance from \( p \) to \( v \), \( d_p(v) \) and a constant to normalize distance values into \([0, 1]\) are sent to the fragment shader. OpenGL rasterizes the segment subdividing it into fragments conforming the discretized edge points. Then, the fragment shader computes the current distance from \( p \) to each fragment by adding to \( d_p(v) \) the distance from the current point to vertex \( v \) which is obtained in the corresponding texture coordinate. This distance is normalized and stored as the fragment depth.

In order to correctly compute a distance function, in the initialization process of Dijkstra’s algorithm we initialize the depth buffer to the maximal depth value \((1)\). When a new fragment is processed, the depth buffer is updated if and only if the depth value of the current fragment is smaller than the
current value in the depth buffer. At the end of the process the value stored in the depth buffer is the minimum depth (distance) defined by all the processed fragments. Consequently the discrete representation of the distance function is obtained during Dijkstra’s algorithm computation.

The time needed to compute and store a distance function in the GPU is $O(N\tau\rho)$, where we denote by $N\tau$ the time needed in Dijkstra’s algorithm to render the image of the edges of $N$ on $R$. Notice that edges are rendered when the distance to one of its vertices is updated, thus each edge is generally rendered more than once. Consequently, the time needed to obtain and store in the GPU a collection of $s$ distance functions is $O(sN\tau\rho)$.

3 Applications

Let $P = \{p_1, \ldots, p_s\}$ be a set of $s$ static sites lying on the road network $N$ and $k \in \{1, \ldots, s\}$. From now on, we assume that: a) each site in $P$ has an identifying index $i = 1, \ldots, s$ and an associated color; b) the discrete distance functions of all the sites have already been computed and are stored in the GPU by using textures; c) each texture stores as much distance functions as possible according to the discretization size and the maximal texture size which depends on the graphics card characteristics. The assumption that all the distance functions are stored in the GPU implies that the maximal number of sites that can be processed depends on the GPU memory amount. However, it notably speeds up the processes and reduces the CPU memory storage.

3.1 Order-$k$ Nearest Neighbor Diagrams

For any point $q \in N$, let $n_k(q)$ denote the $k$-th nearest site to $q$ in $P$, i.e. the site that ranks number $k$ in the ordering of the sites by increasing distance from point $q$. The $k$-th Nearest Neighbor region $N_k(p)$ of a site $p \in P$, is the set of points $q$ of $N$ for which $p$ is the $k$-th nearest site:

$$N_k(p) = \{q \in N \mid n_k(q) = p\}.$$ 

The region $N_k(p)$ is a, possibly unconnected, collection of edges and subedges of $N$.

The Order-$k$ Nearest Neighbor diagram of $P$, denoted $\mathcal{N}_k(P)$, is the partition of $N$ into $k$-th Nearest Neighbor regions:

$$\mathcal{N}_k(P) = \{N_k(p) \mid p \in P\}.$$
The Order-$k$ Nearest Neighbor diagram coincides with the $k$-lower level of the arrangement defined by the distance functions of the sites in $P$.

### 3.1.1 GPU-based Solution

The Order-$k$ Nearest Neighbor diagram can be computed by using the "depth peeling" technique [2]. At each pass all the distance functions are rendered using their corresponding color by defining a rectangle covering $\mathcal{R}$ and associating to its vertices the texture coordinates of the current distance function. By using the depth test and initializing the depth buffer to one, the minimal depth value is stored in the depth buffer. In the first pass the representation of $\mathcal{N}_1(P)$ on $\mathcal{R}$ is obtained in the color buffer. The depth buffer contains the distance from every point of $\mathcal{N}$ to its nearest site. When the first pass finishes, the depth buffer is transferred to a texture. In the second pass all the distance functions are again rendered. In the fragment shader the distance function that is being rendered is compared with the distances obtained in the previous pass. Only the fragments with distance bigger than the one obtained in the previous pass, the others are discarded (by setting them to a depth value bigger than one). Therefore, the distance values obtained in the second pass correspond to the distances to the second nearest site and thus in the color buffer we obtain a representation of $\mathcal{N}_2(P)$ on $\mathcal{R}$. After this process has been repeated $k$ times the representation of $\mathcal{N}_k(P)$ on $\mathcal{R}$ is obtained in the color buffer and the depth buffer stores the distance to the $k$-nearest site.

The representation of $\mathcal{N}_k(P)$ on $\mathcal{N}$ can be obtained by transferring the resulting color information in a texture and rendering $\mathcal{N}$. When edge $e = (v, w)$ is rendered we associate the texture coordinates corresponding to $\tau(e, v)$ and $\tau(e, w)$ to its endpoints so that it uses the color information stored in the texture to color it. Figure 2a shows an Order-7 Nearest Neighbor diagram of 25 sites.

![Figure 2: a) Order-7 Nearest Neighbor diagram; b) Reverse 7-Nearest Neighbor query example.](image)

Computing an Order-$k$ Nearest Neighbor diagram requires rendering the $s$ distance functions $k$ times during the peeling process and, consequently, $O(s k HW \rho)$ time is needed. In order to solve the other problems the Order-$i$ Nearest Neighbor diagrams, $i \leq k$, are stored in the CPU in float arrays, it provides extra $O(k HWt)$ time. Consequently, the total time complexity is $O(s k HW \rho + k HWt)$.

### 3.2 Reverse $k$-Nearest Neighbor Queries

Given a set $P$ of sites and a query point $q$, $P$ and $q$ located on the network, a Reverse $k$-Nearest Neighbor (R-$k$-NN) query retrieves the subset $RNN_k(P, q)$ of sites of $P$ that have $q$ in the set of their $k$-nearest neighbors:

$$RNN_k(P, q) = \{p \in P \mid q \in NN_k(P, p)\}.$$  

For example, a R-$k$-NN query may be issued to find the three stores that are most affected, because of its geographical proximity, by opening a new store at some specific location.
3.2.1 GPU-based Solution

A Reverse $k$-Nearest Neighbor query, $RNN_k(P, q)$, is solved by computing the distance function defined by point $q$ and the distance of each point of $P$ to its $(k+1)$-nearest site. Notice that the distance to the $(k+1)$-nearest site is the value stored in the depth buffer when the Order-$(k+1)$ Nearest Neighbor diagram is obtained. For each site $p_i \in P$ we compare the distance to its $(k+1)$-nearest site, denoted $dn_{k+1}(P, p_i)$, with the distance from $p_i$ to $q$, $d_q(p_i)$. Site $p_i$ is contained in $RNN_k(P, q)$ if and only if $d_q(p_i) \leq dn_{k+1}(P, p_i)$. In Fig. 2b, the solution to a Reverse $7$-Nearest Neighbor for a query point, represented by a circle, is presented in a color gradation according to the ordering of the sites by network distance.

$k$-Reverse Nearest Neighbor queries can be answered in $O(\mathcal{N}_r \rho + H W t + s)$ assuming that we have already computed and stored the corresponding Nearest Network diagram and its distances in the CPU. With this assumptions we only have to compute the distance function $d_q$, store it in the CPU and compare the distance values of the $s$ sites.

3.3 Bichromatic Reverse $k$-Nearest Neighbor Queries

In some applications, the points in $P$ belong to two different categories $B$ and $R$. These two categories may be thought of as being colored blue and red and are usually named sites and points respectively. A Bichromatic Reverse $k$-Nearest Neighbor (BR-$k$-NN) query for a point $r \in R$ seeks to determine the set $BRNN_k(B, R, r)$ of sites $b \in B$ for which $r$ is a $k$-Nearest Neighbor in $R$:

$$BRNN_k(B, R, r) = \{b \in B | r \in NN_k(R, b)\}.$$

For example, given several choices for the location of a new restaurant, a BR-$k$-NN query may be used to evaluate the benefit of the new restaurant in terms of the customers that it may attract from rival restaurants based on the customers living in the five nearest buildings.

3.3.1 GPU-based Solution

In order to solve a Bichromatic $k$-Reverse Nearest Neighbor query, $BRNN_k(B, R, r)$, we first compute the $i$-Nearest Neighbor diagrams of $R$: $N_1(R), \ldots, N_k(R)$. Next for each $b \in B$ we determine whether $r \in NN_k(R, b)$. This is done by checking if $b$ is contained in a $i$-nearest neighbor region of $r$, $N_i(r)$. If $b \in N_i(r)$ with $i \leq k$ then $b \in BRNN_k(R, R, r)$.

The time complexity of a Bichromatic Reverse $k$-Nearest Neighbor query is $O(k |\mathcal{B}|)$, where $|\mathcal{B}|$ denotes the number of blue sites. In this case we assume that we have already computed the $i$-Nearest Network diagrams of the red sites and consequently we provide the time needed to check for each blue site whether it is contained in the desired region.

4 Experimental Evaluation

In this section we present experiments that discuss on the main characteristics of the algorithms presented as well as provide details on their degree of efficiency. We have implemented the proposed methods using C+++, OpenGL and Cg. All the experiments have been carried out on a Intel(R) Core(TM)2 Quad CPU at 2.83GHz with 3.25GB of RAM and a NVIDIA GeForce GTX 280 graphics board. We have used the road network of Nevada with 280,942 nodes and 338,205 edges. In this section we present experiments that show the degree of efficiency of our algorithms. We have worked with TIGER/lines® files that can be found at [4]. In our case, CPU time does not provide meaningful information as most of our calculations are performed in the GPU. Consequently the time presented throughout this section stands for real (calendar) time measurements.

Any graphics board has some specific memory limitations. Our NVIDIA GeForce GTX 280 graphics board has 1G of memory and can work with at most 32 textures simultaneously. Storing the discretized distance functions in the GPU produces that the number of sites that can be fitted in the GPU depends
on the GPU characteristics and the discretization size \((W \times H)\). These two limitations (memory limitation and maximal number of simultaneously storable textures) make an efficient management of the GPU mandatory. We have worked with big textures \((\text{of size } 4096 \times 4096)\) and stored as many discretized distance functions as possible in each of them. For example, for the map of Nevada with \(2048 \times 1024\) resolution, we are able to store up to 256 distance functions.

### 4.1 Distance Function Computation

In this section we present details on some of the fundamental characteristics of our algorithms. First we provide an experimental study on the precision achieved and then we discuss on the algorithm that computes distance functions. In order to highlight the characteristics of the other algorithms, we consider distance function computation as a necessary preprocessing step and only present its computation times in this section.

**Error Analysis**

We can distinguish between two different types of algorithm-related errors: a) Discretization error that depends on the discretization size i.e. the biggest the grid size the smallest the error, and b) Floating point representation errors, which are related to the depth buffer and depth texture precision. A 16-bit precision is sufficient to store the normalized distances which take values in the interval \([0, 1]\).

In the following experiment we aim at quantifying how much does the discretization error contribute to the total error. We have worked with different resolutions for the map of Nevada and calculated the distance between two randomly placed points in the road network using two different methods: a) Software method: We calculate the distance between the two points using Dijkstra’s algorithm with the first point as source; b) Hardware method: We calculate the distance function of the first point and then look up the position of the second point.

The first method suffers only from possible errors in data and of the representation errors related to any algorithm. The second method includes also errors related to the discretization produced by our algorithm and to the precision limitations of GPUs. We compared the distance obtained with each algorithm and obtained a measure of the discretization and GPU precision errors which are only present in the second method. Table 1 presents the name of the maps, the number of nodes in each map, the parametrization resolution, the maximum theoretical error related to the discretization and the mean distance difference observed in 1000 repetitions of the experiment (mean observed error). The maximum theoretical error is calculated as half the length in meters of the side of one of the squares of the discretization grid. Errors are given in meters. The table shows how the use of finer discretization grids grants less theoretical and practical error, it also shows how the precision of our algorithm is high.

**Table 1: Parameterization precision.**

<table>
<thead>
<tr>
<th>Disc. size</th>
<th>Max. err. (m)</th>
<th>Mean err. (m)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024 \times 1024</td>
<td>75.77</td>
<td>37.24</td>
</tr>
<tr>
<td>2048 \times 1024</td>
<td>30.64</td>
<td>19.27</td>
</tr>
<tr>
<td>2048 \times 2048</td>
<td>13.98</td>
<td>7.08</td>
</tr>
</tbody>
</table>

**Table 2: Order-\(k\) Nearest Network diagrams.**

<table>
<thead>
<tr>
<th>(s)</th>
<th>(k = 1)</th>
<th>(k = 10)</th>
<th>(k = 25)</th>
<th>(k = 50)</th>
<th>(k = 100)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.04</td>
<td>0.43</td>
<td>0.91</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>100</td>
<td>0.05</td>
<td>0.53</td>
<td>1.18</td>
<td>2.35</td>
<td>-</td>
</tr>
<tr>
<td>150</td>
<td>0.15</td>
<td>1.56</td>
<td>3.77</td>
<td>7.52</td>
<td>15.03</td>
</tr>
</tbody>
</table>

**Distance Function Computation Times**

In the following we exemplify the amount of time needed to complete the parametrization computation and the process that discretizes distance functions and runs Dijkstra’s algorithm: For the map of Nevada \((n = 280942)\) with a resolution of \(2048 \times 1024\), the computation of the parametrization takes 0.11 seconds and the computation of Dijkstra’s algorithm plus 150 distance functions takes 39.64 seconds. This shows how the parallelization capabilities of graphic cards produce low computational times when computing distance functions, even for maps with a large number of nodes at high resolution. For the rest of experiments we use the map of Nevada with a 2048x1024 resolution.
4.2 Applications

Nearest Network Diagrams

In Table 2 we present the time needed to compute Order-$k$ Nearest Network Diagrams from already computed distance functions.

Reverse Neighbor queries

We used 150 randomly distributed sites in the map of Nevada and performed random $k$-Reverse Nearest Neighbor queries. We executed these queries for several values of $k$ and obtained that, once the times needed for distance function and Nearest Neighbor diagram computations had been subtracted, the time needed to answer the query was a constant 0.55 seconds. $k$-Reverse Nearest Neighbor queries are not affected by $k$ as for each query we only need to compute and read one distance field and can access the desired Order-$i$ Nearest Network diagrams directly.

Bichromatic Reverse Neighbor queries

In order to illustrate the way Bichromatic reverse neighbor queries, first of all we present the time needed for the computation of the necessary Order-$k$ Nearest Network diagrams from already computed distance functions.

Table 3: Bichromatic reverse neighbor queries.

| $k$ | $|B| = 150, |R| = 150$ | $|B| = 50, |R| = 150$ | $|B| = 150, |R| = 50$ |
|-----|-----------------|-----------------|-----------------|
| 10  | 1.58            | 1.59            | 0.45            |
| 20  | 3.14            | 3.14            | 0.87            |
| 30  | 4.60            | 4.60            | 1.19            |
| 40  | 6.26            | 6.26            | 1.72            |
| 50  | 7.71            | 7.70            | 2.03            |

Table 3 presents running times for sets of red and blue points of varying cardinality. We observe that the times presented in the first two columns are almost the same while the time in the third column is much lower. Thus, the time is determined mainly by the number of red points. This happens because for each blue point we only have to decide if it belongs to the Order-$i$ Nearest Network diagrams. Concerning the times needed to solve the Reverse Bichromatic problem once the necessary Order-$k$ Nearest Network diagrams are all around $3.5 \times 10^{-4}$ seconds. Consequently, the time needed to solve the problem is governed by the time needed to compute Order-$k$ Nearest Network diagrams.

References